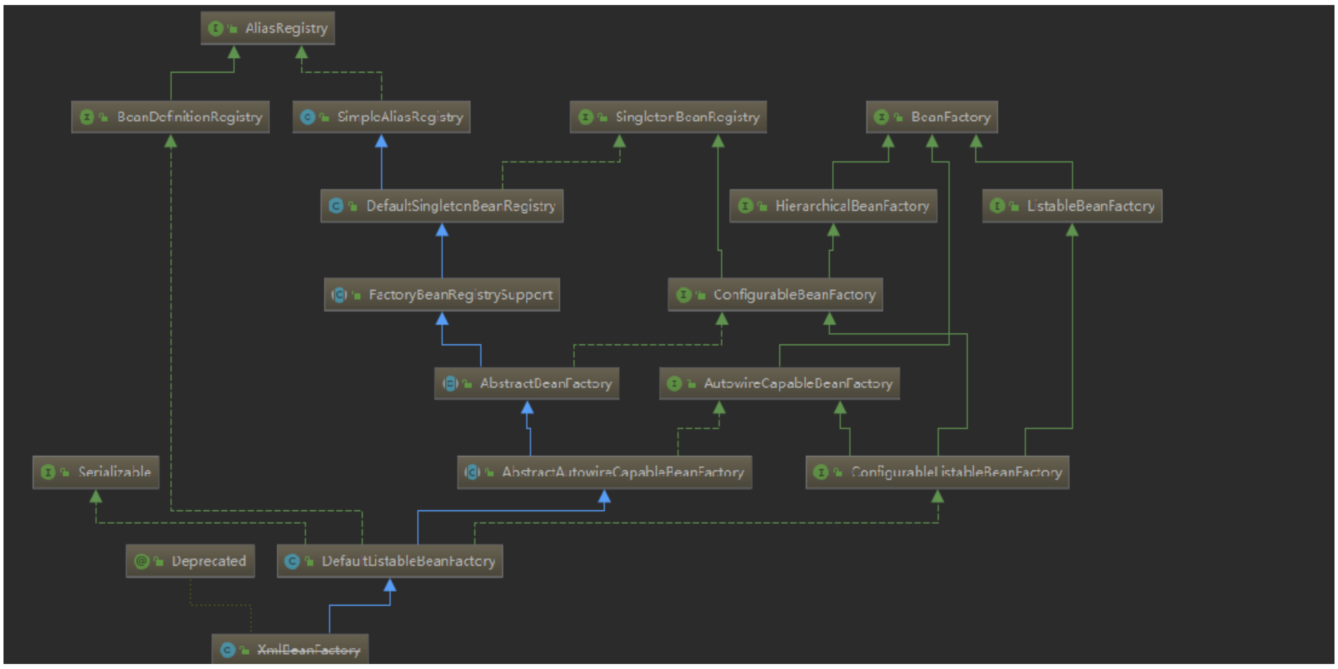
注: 1.阅读前你需要知道DOM解析 2.源码使用的版本(此系列的所有文章都是此版本):

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.3.12.RELEASE</version>
</dependency>
```

XmlBeanFactory在Spring4.0以上已经明确被标示废弃，但为什么还是要解析它呢？ 原因: 1.这是属于spring的元老级别的工厂很多内容后面都需要 2.后面的ApplicationContext中所拿到的所有bean都是通过BeanFactory拿到的,所以不讲不行 3.简单,是了解BeanFactory的最佳切入点

先看几张类的继承关系图



注:BeanFactory的整体继承图

XmlBeanFactory总体来说分两步: 1.bean信息的注册 1.定位 定位bean信息 2.载入并解析 解析bean的配置信息 3.注册 注册到工厂中 2.bean的实例生成 本篇将主要解析bean信息的注册过程

解析入口测试代码:

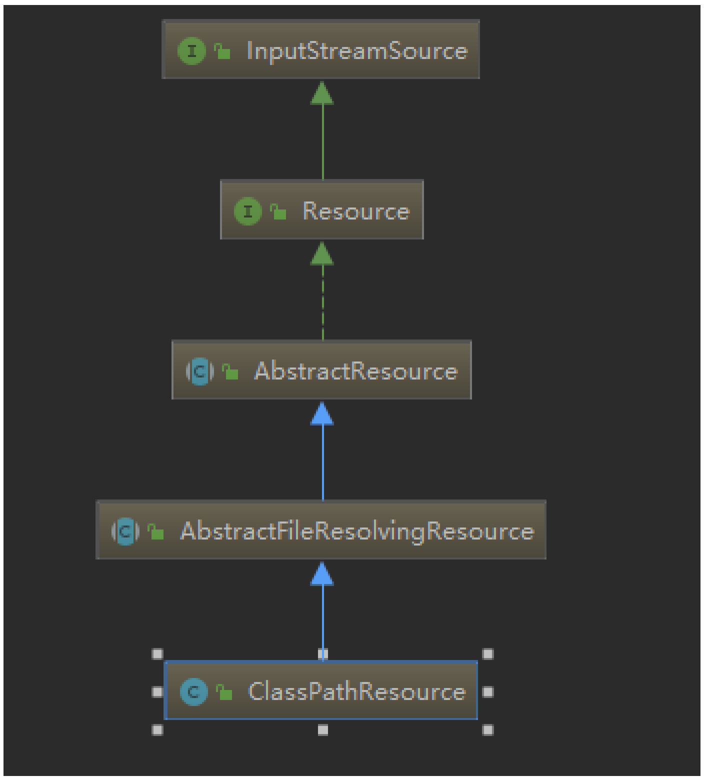
```
ClassPathResource pathResource = new ClassPathResource("/bean.xml");
XmlBeanFactory beanFactory = new XmlBeanFactory(pathResource);
Object person = beanFactory.getBean("person");
```

此处我在配置文件处建了一个bean.xml文件 里面主要配置了一个Perosn类的信息

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="person" class="com.mgw.bean.Person">
        <property name="age" value="10"></property>
        <property name="name" value="sansan"></property>
    </bean>
</beans>
```

解析开始: 先看ClassPathResource这个类



注:ClassPathResource类的继承图

主要主要对配置文件做一个定位

```
public ClassPathResource(String path, ClassLoader classLoader) {
    Assert.notNull(path, "Path must not be null");
    String pathToUse = StringUtils.cleanPath(path);
    //注意此处如果你的测试代码中不是new ClassPathResource("/bean.xml")而是new
ClassPathResource("classpath:/bean.xml")
    //那么它的资源定位可能定位不到,原因就是下面那个if()代码
    if (pathToUse.startsWith("/")) {
        pathToUse = pathToUse.substring(1);
    }
    //将当前配置文件做一个路劲解析并保存
    this.path = pathToUse;
    this.classLoader = (classLoader != null ? classLoader : ClassUtils.getDefaultClassLoader());
}
```

顺便一提,Resource这个接口下的类基本就是做资源定位的作用的，他的子类或者子接口很多，每种略有不同

```java
public interface Resource extends InputStreamSource {

    /**
     * Determine whether this resource actually exists in physical form.
     * <p>This method performs a definitive existence check, whereas the
     * existence of a {@code Resource} handle only guarantees a valid
     * descriptor handle.
     */
    boolean exists();

    /**
     * Indicate whether the contents of this resource can be read via
     * {@link #getInputStream()}.
     * <p>Will be {@code true} for typical resource descriptors;
     * note that actual content reading may still fail when attempted.
     * However, a value of {@code false} is a definitive indication
     * that the resource content cannot be read.
     * @see #getInputStream()
     */
    boolean isReadable();

    /**
     * Indicate whether this resource represents a handle with an open stream.
     * If {@code true}, the InputStream cannot be read multiple times,
     * and must be read and closed to avoid resource leaks.
     * <p>Will be {@code false} for typical resource descriptors.
     */
    boolean isOpen();

    /**
     * Return a URL handle for this resource.
     * @throws IOException if the resource cannot be resolved as URL,
     * i.e. if the resource is not available as descriptor
     */
    URL getURL() throws IOException;

    /**
     * Return a URI handle for this resource.
     * @throws IOException if the resource cannot be resolved as URI,
     * i.e. if the resource is not available as descriptor
     * @since 2.5
     */
    URI getURI() throws IOException;

    /**
     * Return a File handle for this resource.
     * @throws java.io.FileNotFoundException if the resource cannot be resolved as
     * absolute file path, i.e. if the resource is not available in a file system
     * @throws IOException in case of general resolution/reading failures
     * @see #getInputStream()
     */
    File getFile() throws IOException;

    /**
     * Determine the content length for this resource.
     * @throws IOException if the resource cannot be resolved
```

```
	 * (in the file system or as some other known physical resource type)
	 */
	long contentLength() throws IOException;

	/**
	 * Determine the last-modified timestamp for this resource.
	 * @throws IOException if the resource cannot be resolved
	 * (in the file system or as some other known physical resource type)
	 */
	long lastModified() throws IOException;

	/**
	 * Create a resource relative to this resource.
	 * @param relativePath the relative path (relative to this resource)
	 * @return the resource handle for the relative resource
	 * @throws IOException if the relative resource cannot be determined
	 */
	Resource createRelative(String relativePath) throws IOException;

	/**
	 * Determine a filename for this resource, i.e. typically the last
	 * part of the path: for example, "myfile.txt".
	 * <p>Returns {@code null} if this type of resource does not
	 * have a filename.
	 */
	String getFilename();

	/**
	 * Return a description for this resource,
	 * to be used for error output when working with the resource.
	 * <p>Implementations are also encouraged to return this value
	 * from their {@code toString} method.
	 * @see Object#toString()
	 */
	String getDescription();

}
```

BeanDefinitionRegistry接口 这个接口及其实现类主要是用来操作bean信息的 例如:bean信息的注册 删除 获取等

```
public interface BeanDefinitionRegistry extends AliasRegistry {

	/**
	 * Register a new bean definition with this registry.
	 * Must support RootBeanDefinition and ChildBeanDefinition.
	 * @param beanName the name of the bean instance to register
	 * @param beanDefinition definition of the bean instance to register
	 * @throws BeanDefinitionStoreException if the BeanDefinition is invalid
	 * or if there is already a BeanDefinition for the specified bean name
	 * (and we are not allowed to override it)
	 * @see RootBeanDefinition
	 * @see ChildBeanDefinition
	 */
	void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
			throws BeanDefinitionStoreException;
```

```
    /**
     * Remove the BeanDefinition for the given name.
     * @param beanName the name of the bean instance to register
     * @throws NoSuchBeanDefinitionException if there is no such bean definition
     */
    void removeBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;

    /**
     * Return the BeanDefinition for the given bean name.
     * @param beanName name of the bean to find a definition for
     * @return the BeanDefinition for the given name (never {@code null})
     * @throws NoSuchBeanDefinitionException if there is no such bean definition
     */
    BeanDefinition getBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;

    /**
     * Check if this registry contains a bean definition with the given name.
     * @param beanName the name of the bean to look for
     * @return if this registry contains a bean definition with the given name
     */
    boolean containsBeanDefinition(String beanName);

    /**
     * Return the names of all beans defined in this registry.
     * @return the names of all beans defined in this registry,
     * or an empty array if none defined
     */
    String[] getBeanDefinitionNames();

    /**
     * Return the number of beans defined in the registry.
     * @return the number of beans defined in the registry
     */
    int getBeanDefinitionCount();

    /**
     * Determine whether the given bean name is already in use within this registry,
     * i.e. whether there is a local bean or alias registered under this name.
     * @param beanName the name to check
     * @return whether the given bean name is already in use
     */
    boolean isBeanNameInUse(String beanName);

}
```

XmlBeanFactory这个类

```
public class XmlBeanFactory extends DefaultListableBeanFactory {

    //直接用XmlBeanDefinitionReader这个类及其父类做bean文件的解析 注意这里传入的this
    private final XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(this);


    /**
     * Create a new XmlBeanFactory with the given resource,
     * which must be parsable using DOM.
     * @param resource XML resource to load bean definitions from
     * @throws BeansException in case of loading or parsing errors
     */
    public XmlBeanFactory(Resource resource) throws BeansException {
        this(resource, null);
    }

    /**
     * Create a new XmlBeanFactory with the given input stream,
     * which must be parsable using DOM.
     * @param resource XML resource to load bean definitions from
     * @param parentBeanFactory parent bean factory
     * @throws BeansException in case of loading or parsing errors
     */
    public XmlBeanFactory(Resource resource, BeanFactory parentBeanFactory) throws BeansException
{
        super(parentBeanFactory);
        //将resource资源传入，使用reader进行载入bean并且解析
        this.reader.loadBeanDefinitions(resource);
    }

}
```

步骤； 0.准备工作 上面的new XmlBeanDefinitionReader(this) 注意他传入的this 他是作为BeanDefinitionRegistry这个类注入的，前面我们看XmlBeanFactory的类关系图，他确实实现了BeanDefinitionRegistry这个接口，这里很重要。 它将其传给它的父类AbstractBeanDefinitionReader中

```
protected AbstractBeanDefinitionReader(BeanDefinitionRegistry registry) {
    Assert.notNull(registry, "BeanDefinitionRegistry must not be null");
    //将Registry注册器保存起来,待后面注册时使用
    this.registry = registry;

    // Determine ResourceLoader to use.
    if (this.registry instanceof ResourceLoader) {
        this.resourceLoader = (ResourceLoader) this.registry;
    }
    else {
        this.resourceLoader = new PathMatchingResourcePatternResolver();
    }

    // Inherit Environment if possible
    if (this.registry instanceof EnvironmentCapable) {
        this.environment = ((EnvironmentCapable) this.registry).getEnvironment();
    }
    else {
        this.environment = new StandardEnvironment();
    }
}
```

下面开始载入资源:this.reader.loadBeanDefinitions(resource) 开始 1.创建一个set用来保存正在被载入的资源(保证每一个资源都被载入并解析过)

2.开始做资源载入(因为bean.xml资源文件说到底还是一个文件所以使用IO流来做读入)

3.将资源转成一个Document对象，留待后续使用

```java
public int loadBeanDefinitions(EncodedResource encodedResource) throws
BeanDefinitionStoreException {
    Assert.notNull(encodedResource, "EncodedResource must not be null");
    if (logger.isInfoEnabled()) {
        logger.info("Loading XML bean definitions from " + encodedResource.getResource());
    }
    //创建一个set用来保存正在被载入的资源(保证每一个资源都被载入并解析过)
    Set<EncodedResource> currentResources = this.resourcesCurrentlyBeingLoaded.get();
    if (currentResources == null) {
        currentResources = new HashSet<EncodedResource>(4);
        this.resourcesCurrentlyBeingLoaded.set(currentResources);
    }
    if (!currentResources.add(encodedResource)) {
        throw new BeanDefinitionStoreException(
                "Detected cyclic loading of " + encodedResource + " - check your import
definitions!");
    }
    try {
        //开始做资源载入(因为bean.xml资源文件说到底还是一个文件所以使用IO流来做读入)
        InputStream inputStream = encodedResource.getResource().getInputStream();
        try {
            InputSource inputSource = new InputSource(inputStream);
            if (encodedResource.getEncoding() != null) {
                inputSource.setEncoding(encodedResource.getEncoding());
            }
            //做资源载入
            return doLoadBeanDefinitions(inputSource, encodedResource.getResource());
        }
        finally {
            inputStream.close();
        }
    }
    catch (IOException ex) {
        throw new BeanDefinitionStoreException(
                "IOException parsing XML document from " + encodedResource.getResource(), ex);
    }
    finally {
        currentResources.remove(encodedResource);
        if (currentResources.isEmpty()) {
            this.resourcesCurrentlyBeingLoaded.remove();
        }
    }
}
```

3.将资源转成一个Document对象，留待后续使用

```java
//doLoadBeanDefinitions() 做资源载入
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
            throws BeanDefinitionStoreException {
    try {
        //将资源转成一个Document对象，留待后续使用
        Document doc = doLoadDocument(inputSource, resource);
        //解析xml信息并注册资源
        return registerBeanDefinitions(doc, resource);
    }
    catch (BeanDefinitionStoreException ex) {
        throw ex;
    }
    catch (SAXParseException ex) {
        throw new XmlBeanDefinitionStoreException(resource.getDescription(),
                "Line " + ex.getLineNumber() + " in XML document from " + resource + " is 
invalid", ex);
    }
    catch (SAXException ex) {
        throw new XmlBeanDefinitionStoreException(resource.getDescription(),
                "XML document from " + resource + " is invalid", ex);
    }
    catch (ParserConfigurationException ex) {
        throw new BeanDefinitionStoreException(resource.getDescription(),
                "Parser configuration exception parsing XML from " + resource, ex);
    }
    catch (IOException ex) {
        throw new BeanDefinitionStoreException(resource.getDescription(),
                "IOException parsing XML document from " + resource, ex);
    }
    catch (Throwable ex) {
        throw new BeanDefinitionStoreException(resource.getDescription(),
                "Unexpected exception parsing XML document from " + resource, ex);
    }
}
```

4.得到BeanDefinitionDocumentReader用其解析资源

```
public int registerBeanDefinitions(Document doc, Resource resource) throws 
BeanDefinitionStoreException {
    //创建一个一个BeanDefinitionDocumentReader  就是用来读Document的
    BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();
    //查看之前已经注册好的BeanDefinition信息
    int countBefore = getRegistry().getBeanDefinitionCount();
    //解析并注册资源实际上是调用他的父类(DefaultBeanDefinitionDocumentReader)完成的
    //此处创建了一个ReaderContext(XmlReaderContext)上下文读取对象，并将资源传入(这么做实际上就是对资源进行一个
包装),spring经常做个xxxContext就是因为好用
    documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
    return getRegistry().getBeanDefinitionCount() - countBefore;
}
//ReaderContext的创建此处需要特别注意
public XmlReaderContext createReaderContext(Resource resource) {
    return new XmlReaderContext(resource, this.problemReporter, this.eventListener,
                                this.sourceExtractor, this, getNamespaceHandlerResolver());
}
//特别注意此处把reader(XmlBeanDefinitionReader)这个实例保存到了ReaderContext这个上下文对象了
public XmlReaderContext(
            Resource resource, ProblemReporter problemReporter,
            ReaderEventListener eventListener, SourceExtractor sourceExtractor,
            XmlBeanDefinitionReader reader, NamespaceHandlerResolver namespaceHandlerResolver) {

    super(resource, problemReporter, eventListener, sourceExtractor);
    this.reader = reader;
    this.namespaceHandlerResolver = namespaceHandlerResolver;
}
注:最后解析出来的bean信息都会包装成BeanDefinition进行使用
```

```
此处引出最后保存BeanDefinition的地方为一个Map，注意这个Map是被包装的线程安全的HashMap，其为何会线程安全后面会专门开一个专题来讲java的多线程，此处略过
/** Map of bean definition objects, keyed by bean name */
private final Map<String, BeanDefinition> beanDefinitionMap = new ConcurrentHashMap<String, BeanDefinition>(256);
```

5. DefaultBeanDefinitionDocumentReader类中继续解析xml文件以及完成后续的注册

```
public void registerBeanDefinitions(Document doc, XmlReaderContext readerContext) {
    //保存readerContext这个读取器上线文
    this.readerContext = readerContext;
    logger.debug("Loading bean definitions");
    //将Document解析成一个root标签
    Element root = doc.getDocumentElement();
    //从root标签开始解析xml文件并注册
    doRegisterBeanDefinitions(root);
}
顺便一说DefaultBeanDefinitionDocumentReader这个类  看这个类中定义的一个常量,全是spring配置文件的一些标签节点类型
public class DefaultBeanDefinitionDocumentReader implements BeanDefinitionDocumentReader {

    public static final String BEAN_ELEMENT = BeanDefinitionParserDelegate.BEAN_ELEMENT;

    public static final String NESTED_BEANS_ELEMENT = "beans";

    public static final String ALIAS_ELEMENT = "alias";

    public static final String NAME_ATTRIBUTE = "name";

    public static final String ALIAS_ATTRIBUTE = "alias";

    public static final String IMPORT_ELEMENT = "import";

    public static final String RESOURCE_ATTRIBUTE = "resource";

    public static final String PROFILE_ATTRIBUTE = "profile";
}
```

6.创建xml节点标签解析的代表器用来解析xml文件

```
protected void doRegisterBeanDefinitions(Element root) {
    // Any nested <beans> elements will cause recursion in this method. In
    // order to propagate and preserve <beans> default-* attributes correctly,
    // keep track of the current (parent) delegate, which may be null. Create
    // the new (child) delegate with a reference to the parent for fallback purposes,
    // then ultimately reset this.delegate back to its original (parent) reference.
    // this behavior emulates a stack of delegates without actually necessitating one.
    BeanDefinitionParserDelegate parent = this.delegate;
    //创建一个BeanDefinitionParserDelegate用来解析xml中标签节点
    this.delegate = createDelegate(getReaderContext(), root, parent);

    if (this.delegate.isDefaultNamespace(root)) {
        String profileSpec = root.getAttribute(PROFILE_ATTRIBUTE);
        if (StringUtils.hasText(profileSpec)) {
            String[] specifiedProfiles = StringUtils.tokenizeToStringArray(
                    profileSpec,
BeanDefinitionParserDelegate.MULTI_VALUE_ATTRIBUTE_DELIMITERS);
            if (!getReaderContext().getEnvironment().acceptsProfiles(specifiedProfiles)) {
                if (logger.isInfoEnabled()) {
                    logger.info("Skipped XML bean definition file due to specified profiles ["
+ profileSpec +
                            "] not matching: " + getReaderContext().getResource());
                }
                return;
            }
        }
    }
    //解析xml前的准备工作,实际上是个空方法,spring希望如果用户自己实现BeanDefinitionDocumentReader的子类
时你可以重写这个方法这是spring扩展性的延伸, 在后面的一些其他源码解析中你可以看到大量的这样使用
    preProcessXml(root);
    //开始做解析
    parseBeanDefinitions(root, this.delegate);
    //解析xml并注册完成后的一些后置工作,实际上是个空方法,也是希望如果你自己实现了
BeanDefinitionDocumentReader子类你可以重写
    postProcessXml(root);

    this.delegate = parent;
}
//开始做解析
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
    if (delegate.isDefaultNamespace(root)) {
        NodeList nl = root.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (node instanceof Element) {
                //判断是Element类型的开始解析
                Element ele = (Element) node;
                if (delegate.isDefaultNamespace(ele)) {
                    parseDefaultElement(ele, delegate);
                }
                else {
                    delegate.parseCustomElement(ele);
                }
            }
        }
    }
    else {
        delegate.parseCustomElement(root);
    }
}
```

7.开始解析Element

```
private void parseDefaultElement(Element ele, BeanDefinitionParserDelegate delegate) {
    if (delegate.nodeNameEquals(ele, IMPORT_ELEMENT)) {
        //解析import类型标签
        importBeanDefinitionResource(ele);
    }
    else if (delegate.nodeNameEquals(ele, ALIAS_ELEMENT)) {
        //解析alias类型节点
        processAliasRegistration(ele);
    }
    else if (delegate.nodeNameEquals(ele, BEAN_ELEMENT)) {
        //解析bean类型节点
        processBeanDefinition(ele, delegate);
    }
    else if (delegate.nodeNameEquals(ele, NESTED_BEANS_ELEMENT)) {
        // recurse
        //解析beans类型节点 实际上就是调用刚刚上面的doRegisterBeanDefinitions方法
        doRegisterBeanDefinitions(ele);
    }
}
//解析bean类型节点
protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {
    BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
    if (bdHolder != null) {
        bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
        try {
            // Register the final decorated instance.
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext().getRegistry());
        }
        catch (BeanDefinitionStoreException ex) {
            getReaderContext().error("Failed to register bean definition with name '" +
                    bdHolder.getBeanName() + "'", ele, ex);
        }
        // Send registration event.
        getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));
    }
}
```

8.解析Bean标签

```
protected void processBeanDefinition(Element ele, BeanDefinitionParserDelegate delegate) {
    //调用前面已经创建好的BeanDefinitionParserDelegate代表器来解析bean标签，然后保存到
BeanDefinitionHolder这个里面  这个里面有BeanDefinition信息
    //具体怎么解析就略过，前面开头时已经说过了你需要DOM解析的相关知识
    BeanDefinitionHolder bdHolder = delegate.parseBeanDefinitionElement(ele);
    if (bdHolder != null) {
        bdHolder = delegate.decorateBeanDefinitionIfRequired(ele, bdHolder);
        try {
            // Register the final decorated instance.
            //从Read上下文中得到Registry注册器  注意此时才到我们的第三阶段BeanDefinition的注册,前面的所
有都只是在载入解析bean信息而已
            //使用BeanDefinitionReaderUtils进行bean信息的注册
            BeanDefinitionReaderUtils.registerBeanDefinition(bdHolder,
getReaderContext().getRegistry());
        }
        catch (BeanDefinitionStoreException ex) {
            getReaderContext().error("Failed to register bean definition with name '" +
                    bdHolder.getBeanName() + "'", ele, ex);
        }
        // Send registration event.
        getReaderContext().fireComponentRegistered(new BeanComponentDefinition(bdHolder));
    }
}
此处重点说明下上面的getReaderContext().getRegistry()为什么可以得到一个注册器
```

```
原因:
我们在步骤0准备工作哪里说了XmlBeanFactory这个类，他实现了BeanDefinitionRegistry这个接口虽然你在
XmlBeanFactory这个类中找不到接口的实现方法，
但是你可以在他的父类中找到接口的实现方法，并且在XmlbeanFactory中的
->private final XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(this);这句代码将其
赋给了XmlBeanDefinitionReader的父类并保存起来了
然后在步骤4中我们将reader又保存到了ReadContext中了
最后在步骤5中我们将readerContext保存起来了所以可以通过getReaderContext().getRegistry()
                                                            -->return
this.reader.getRegistry();
得到注册器，注意这个注册器实际上就是XmlBeanFactory的父类DefaultListableBeanFactory他实现了
BeanDefinitionRegistry接口的全部方法
public interface BeanDefinitionRegistry extends AliasRegistry {
    void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
            throws BeanDefinitionStoreException;
    void removeBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;
    BeanDefinition getBeanDefinition(String beanName) throws NoSuchBeanDefinitionException;
    boolean containsBeanDefinition(String beanName);
    String[] getBeanDefinitionNames();
    int getBeanDefinitionCount();
    boolean isBeanNameInUse(String beanName);

}
public class DefaultListableBeanFactory extends AbstractAutowireCapableBeanFactory
        implements ConfigurableListableBeanFactory, BeanDefinitionRegistry, Serializable {
    ...
    ...
    ...
}
```

9.注册BeanDefinition

```
public static void registerBeanDefinition(
        BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry registry)
        throws BeanDefinitionStoreException {

    // Register bean definition under primary name.
    String beanName = definitionHolder.getBeanName();
    //此处回调注册器(DefaultListableBeanFactory)的registerBeanDefinition方法
    registry.registerBeanDefinition(beanName, definitionHolder.getBeanDefinition());

    // Register aliases for bean name, if any.
    String[] aliases = definitionHolder.getAliases();
    if (aliases != null) {
        for (String alias : aliases) {
            registry.registerAlias(beanName, alias);
        }
    }
}
```

10.回调方法注册BeanDefinition

```
public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
            throws BeanDefinitionStoreException {
```

```
    Assert.hasText(beanName, "Bean name must not be empty");
    Assert.notNull(beanDefinition, "BeanDefinition must not be null");

    if (beanDefinition instanceof AbstractBeanDefinition) {
        try {
            ((AbstractBeanDefinition) beanDefinition).validate();
        }
        catch (BeanDefinitionValidationException ex) {
            throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(),
beanName,
                    "Validation of bean definition failed", ex);
        }
    }

    BeanDefinition oldBeanDefinition;
    //判断这个要添加进来的bean信息是否已经存在了
    oldBeanDefinition = this.beanDefinitionMap.get(beanName);
    if (oldBeanDefinition != null) {
        //如果存在了判断各种条件看是否需要更新,如果有一个条件不满足则直接抛异常，条件都满足后才实行更新
        if (!isAllowBeanDefinitionOverriding()) {
            throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(),
beanName,
                    "Cannot register bean definition [" + beanDefinition + "] for bean '" +
beanName +
                    "': There is already [" + oldBeanDefinition + "] bound.");
        }
        else if (oldBeanDefinition.getRole() < beanDefinition.getRole()) {
            // e.g. was ROLE_APPLICATION, now overriding with ROLE_SUPPORT or
ROLE_INFRASTRUCTURE
            if (this.logger.isWarnEnabled()) {
                this.logger.warn("Overriding user-defined bean definition for bean '" +
beanName +
                        "' with a framework-generated bean definition: replacing [" +
                        oldBeanDefinition + "] with [" + beanDefinition + "]");
            }
        }
        else if (!beanDefinition.equals(oldBeanDefinition)) {
            if (this.logger.isInfoEnabled()) {
                this.logger.info("Overriding bean definition for bean '" + beanName +
                        "' with a different definition: replacing [" + oldBeanDefinition +
                        "] with [" + beanDefinition + "]");
            }
        }
        else {
            if (this.logger.isDebugEnabled()) {
                this.logger.debug("Overriding bean definition for bean '" + beanName +
                        "' with an equivalent definition: replacing [" + oldBeanDefinition +
                        "] with [" + beanDefinition + "]");
            }
        }
        this.beanDefinitionMap.put(beanName, beanDefinition);
    }
    else {
        //hasBeanCreationStarted()判断当前bean是否被被多次注册，限制多线程情况下的多次注册
        if (hasBeanCreationStarted()) {
            // Cannot modify startup-time collection elements anymore (for stable iteration)
            synchronized (this.beanDefinitionMap) {
                //注册beanDefinition ,beanDefinitionMap将beanDefinition加入  前面已经说过实际上是
beanDefinitionMap这个Map注册里放了beanDefinition的信息
                this.beanDefinitionMap.put(beanName, beanDefinition);
                List<String> updatedDefinitions = new ArrayList<String>
(this.beanDefinitionNames.size() + 1);
                updatedDefinitions.addAll(this.beanDefinitionNames);
                updatedDefinitions.add(beanName);
                this.beanDefinitionNames = updatedDefinitions;
                if (this.manualSingletonNames.contains(beanName)) {
                    Set<String> updatedSingletons = new LinkedHashSet<String>
(this.manualSingletonNames);
                    updatedSingletons.remove(beanName);
                    this.manualSingletonNames = updatedSingletons;
                }
            }
        }
        else {
            // Still in startup registration phase
            //注册beanDefinition
            this.beanDefinitionMap.put(beanName, beanDefinition);
            this.beanDefinitionNames.add(beanName);
            this.manualSingletonNames.remove(beanName);
        }
        this.frozenBeanDefinitionNames = null;
    }

    if (oldBeanDefinition != null || containsSingleton(beanName)) {
        resetBeanDefinition(beanName);
    }
}
```

以上10个步骤就是整个bean信息被解析成BeanDefinition并注册进工厂的过程 总结: 1.XmlBeanFactory首先是一个bean工厂还是一个注册器BeanDefinitionRegistry 2.bean的相关信息(懒加载，单例还是多例等)全都会被保存到BeanDefinition，并且保存到bean工厂中 3.实际的相关调用关系如下: ClassPathResource用来定位bean资源的位置信息 XmlBeanFactory建立一个XmlBeanDefinitionReader实例用它来载入bean信息 XmlBeanDefinitionReader创建BeanDefinitionDocumentReader实例用来读取Document信息 BeanDefinitionDocumentReader创建BeanDefinitionParserDelegate实例用来接卸xml的节点标签等信息 BeanDefinitionReaderUtils工具类回调注册器的注册方法将bean信息放入工厂中